2INC0 - Operating Systems

Virtual Memory
Part 1

Geoffrey Nelissen

Interconnected
Resource-aware
Intelligent Systems

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Course Overview

- **Introduction to operating systems** (lecture 1)
- **Processes, threads and scheduling** (lectures 2+3)
- **Concurrency and synchronization**
  - atomicity and interference (lecture 4)
  - actions synchronization (lecture 5)
  - condition synchronization (lecture 6)
  - deadlock (lecture 7)
- **File systems** (lecture 8)
- **Memory management (lectures 9+10)**
- **Input/output** (lecture 11)

One of the most important subsystems in an OS. It impacts:
- **speed** of execution
- **Maximum size** of executable programs
- **how many** processes can be executed concurrently
- how data and code can be shared
- …

Memory management

***Program$_1$*:**

```
int x=0;
void main(){
  int x=0;
  while (true) {
        x--;
        if( x > 8 ) {
          something(x);
        } else {
      func();
    }
  }
}
```

compilation

```
Code                                Mem address

        …                           …
        sll  $t1, $s3, 2            0x1000A800
        add  $t1, $t1, $s6          0x1000A804
        …                           …
        lw   $a0, x                 0x1000AFA0
        jal  Proc                   0x1000AFA4
        addi $s3, $s3, 1            0x1000AFA8
        …                           …
        …                           …
        …                           …
Data
        x                           0x1000FF44
```

The compiler/linker associate a (**logical**) address to every instruction and data

Processes creation

Process 1

```
Code

        …
        sll  $t1, $s3, 2
        add  $t1, $t1, $s6
        …
        lw   $a0, x
        jal  Proc
        addi $s3, $s3, 1
        …
        …
        …
Data
        x    0x1000FF44
```

Process 2

```
Code

        …
        sll  $t1, $s3, 2
        add  $t1, $t1, $s6
        …
        lw   $a0, x
        jal  Proc
        addi $s3, $s3, 1
        …
        …
        …
Data
        x    0x1000FF44
```

How do we differentiate between x in Process 1 and Process 2 if they have the same address?

Memory management

***Program$_1$*:**

```
int x=0;
void main(){
   int x=0;
   while (true) {
            x--;
            if( x > 8 ) {
              something(x);
            } else {
         func();
      }
    }
}
```

compilation

Code	Mem address
…	…
sll $t1, $s3, 2	0x1000A800
add $t1, $t1, $s6	0x1000A804
…	…
lw $a0, x	0x1000AFA0
jal Proc	0x1000AFA4
addi $s3, $s3, 1	0x1000AFA8
…	…
…	…
…	…
Data	
x	0x0000FF44

The compiler/linker associate a (**logical**) address to every instruction and data

Processes creation

Process 1

```
Code
      …
      sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      …
      lw   $a0, x
      jal  Proc
      addi $s3, $s3, 1
      …
      …
      …
Data
      x      0x0000FF44
```

Process 2

```
Code
      …
      sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      …
      lw   $a0, x
      jal  Proc
      addi $s3, $s3, 1
      …
      …
      …
Data
      x      0x0000FF44
```

Process loading in main memory

The **physical** address of x in main memory is different

Physical memory

0

```
…
sll  $t1, $s3, 2
add  $t1, $t1, $s6
…
lw   $a0, x
jal  Proc
addi $s3, $s3, 1
…
…
…
x     0x0000FF44
```

300K

388K

```
…
sll  $t1, $s3, 2
add  $t1, $t1, $s6
…
lw   $a0, x
jal  Proc
addi $s3, $s3, 1
…
…
…
x     0x00070F44
```

688K

Memory management

***Program$_1$*:**

```
int x=0;
void main(){
  int x=0;
  while (true) {
        x--;
        if( x > 8 ) {
          something(x);
        } else {
      func();
    }
  }
}
```

compilation

Code	Mem address
…	…
sll $t1, $s3, 2	0x1000A800
add $t1, $t1, $s6	0x1000A804
…	…
lw $a0, x	0x1000AFA0
jal Proc	0x1000AFA4
addi $s3, $s3, 1	0x1000AFA8
…	…
…	…
…	…
Data	
x	0x0000FF44

The compiler/linker associate a (**logical**) address to every instruction and data

Processes creation

Process 1

```
Code
    …
    sll  $t1, $s3, 2
    add  $t1, $t1, $s6
    …
    lw   $a0, x
    jal  Proc
    addi $s3, $s3, 1
    …
    …
    …
Data
    x    0x0000FF44
```

Process 2

```
Code
    …
    sll  $t1, $s3, 2
    add  $t1, $t1, $s6
    …
    lw   $a0, x
    jal  Proc
    addi $s3, $s3, 1
    …
    …
    …
Data
    x    0x0000FF44
```

Physical memory

0

```
…
sll  $t1, $s3, 2
add  $t1, $t1, $s6
…
lw   $a0, x
        1
        44
```

300K

388K

```
…
sll  $t1, $s3, 2
add  $t1, $t1, $s6
…
lw   $a0, x
jal  Proc
addi $s3, $s3, 1
…
…
…
x    0x00070F44
```

688K

➔ the memory management subsystem decides **what to load in main memory, when and where**

Process loading in main memory

The **physical** address of x in main memory is different

➔ the memory management subsystem **translates logical addresses to physical addresses** for each process in main memory

Agenda

- Reminder
- Memory paging
- Conclusion

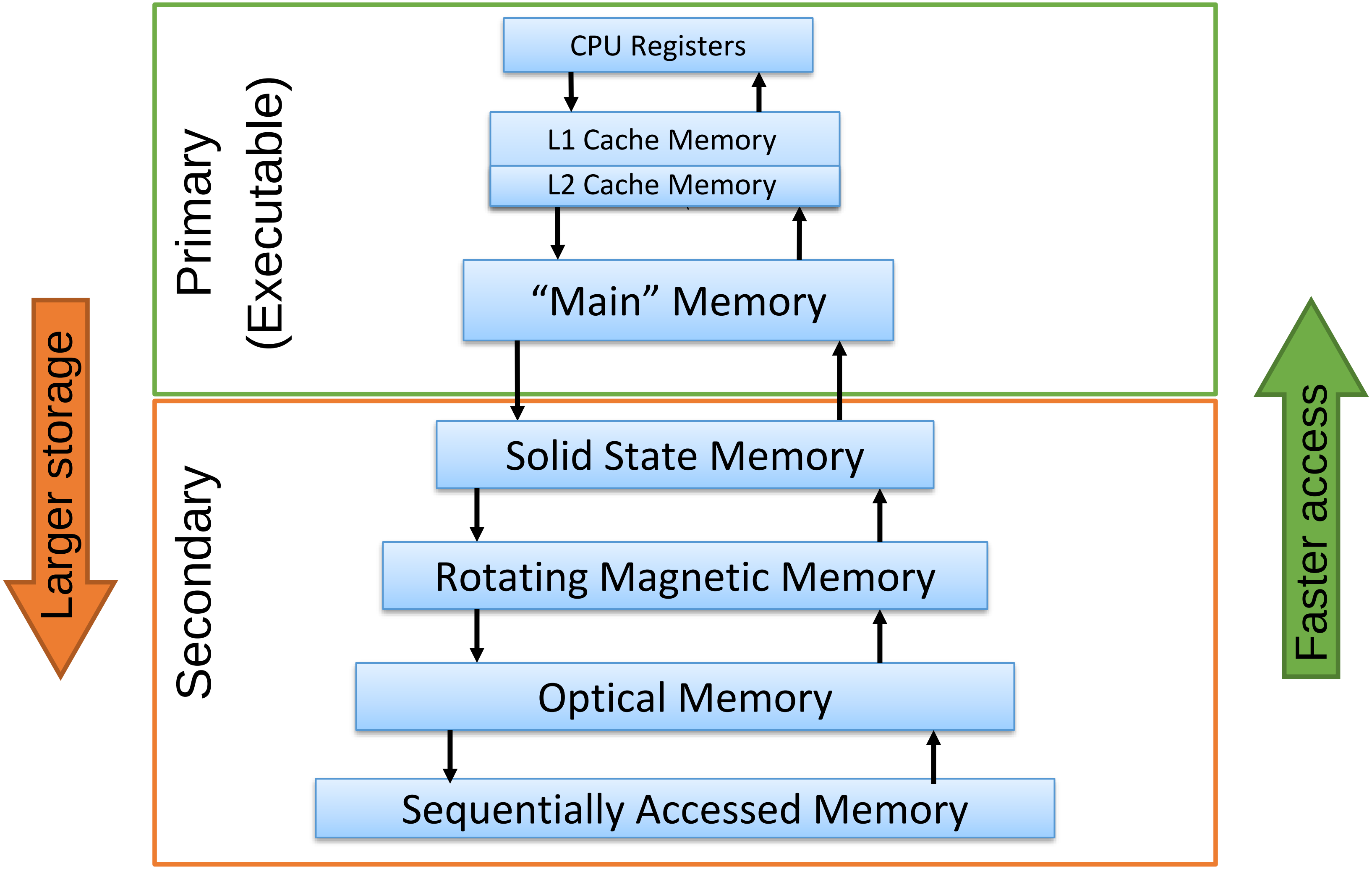
Expectation from an ideal memory

What would we expect from an **ideal memory**?

- Simple to use
  - The programmer should not have to know where data and code are loaded
- Private (isolation)
  - Tasks should be able to share only what they want to
- Non-volatile / permanent
  - Data and code should always be available in memory
- Fast access
  - For better performance
- Huge (unlimited) capacity
  - To be able to execute large programs and treat lots of data
- Cheap
  - To be cost-effective

Conflicting requirements!
We use a **memory hierarchy** and **virtualization** to find a good **tradeoff**.

Memory Hierarchy



Memory management in early systems

- Used to use **memory partitioning**
- Every active process resides **in its entirety in main memory**
  - **Large programs cannot execute**
- Every active process is **allocated a contiguous part of main memory**.

See preparatory material

Memory management in early systems

See preparatory material

- Every active process resides **in its entirety in main memory**
  - **Large programs cannot execute**
- Every active process is **allocated a contiguous part of main memory**.

- Three **partitioning schemes**

Fixed

- **Partitions** have **fixed size**
- Processes are **assigned to a free partition** that is big enough

Dynamic

Relocatable

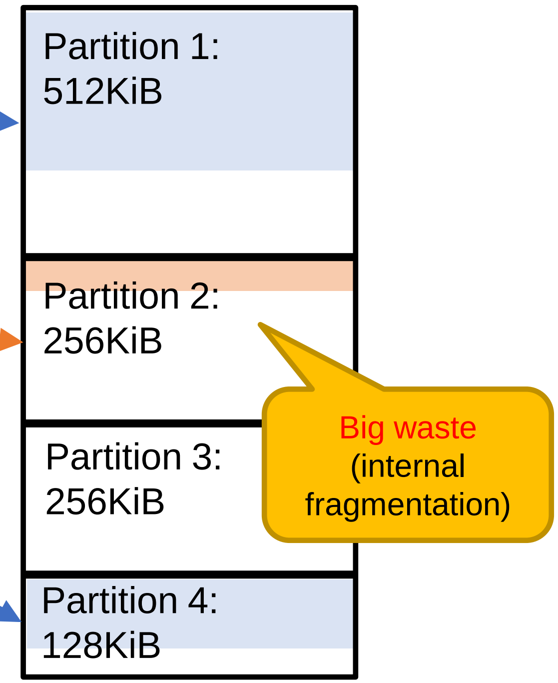
Fixed partitioning: example

Assume a main memory partitioned as shown below, and a process to partition mapping using **Best Fit**.
Consider the following **sequence of process arrivals**, and the size of their memory address space.

- P1: 300KiB
- P2: 78KiB
- P3: 512KiB
- P4: 32KiB
- P5: 600KiB

P3 cannot be loaded until P1 is swapped out, even though there is enough free space

P5 will never be able to execute because it is too big



Memory management in early systems

See preparatory material

- Every active process resides **in its entirety in main memory**
  - **Large programs cannot execute**
- Every active process is **allocated a contiguous part of main memory**.

- Three **partitioning schemes**

Fixed

- **Partitions** have **fixed size**
- Processes are **assigned to a free partition** that is big enough
- **Number of active processes** limited by **number of partitions**
- **Size of processes** limited by the **size of** the largest **partition**
- Wastes memory due to **internal fragmentation**

Dynamic

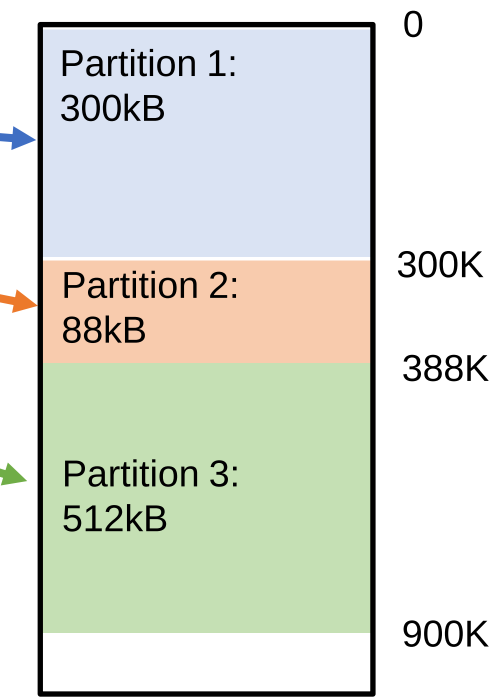
- Processes are **dynamically allocated** space in main memory

Relocatable

Dynamic partitioning: example

Assume a main memory of 1MB as shown below, and a process to partition mapping using **Best Fit**.
Consider the following **sequence of process arrivals and leavings**, and the size of their memory address space.

- P1: 300kB
- P2: 88kB
- P3: 512kB
- P2 completes and releases memory
- P4: 150kB
- P5: 1MB
- P6: 2MB



Dynamic partitioning: example

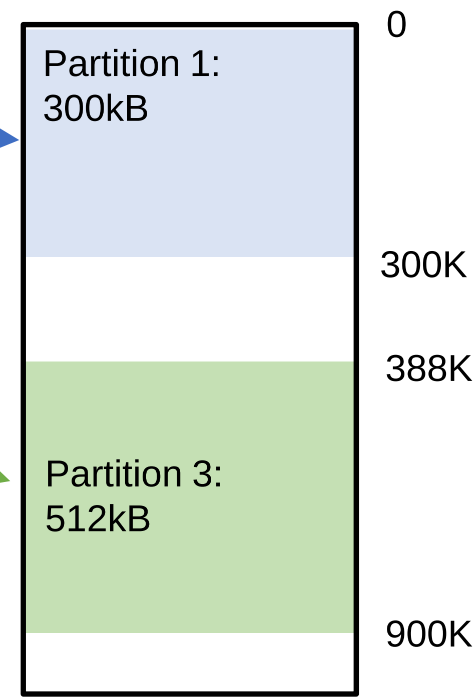
Assume a main memory of 1MB as shown below, and a process to partition mapping using **Best Fit**.
Consider the following **sequence of process arrivals and leavings**, and the size of their memory address space.

- P1: 300kB
- P2: 88kB
- P3: 512kB
- P2 completes and release memory
- P4: 150kB
- P5: 1MB
- P6: 2MB

P4 cannot be loaded even though there is enough free space (external fragmentation)

P5 requires exclusive access to the main memory

P6 will never be able to execute because it is too big



Memory management in early systems

- Every active process resides **in its entirety in main memory**
  - **Large programs cannot execute**
- Every active process is **allocated a contiguous part of main memory**.

See preparatory material

- Three **partitioning schemes**

Fixed

- **Partitions** have **fixed size**
- Processes are **assigned to a free partition** that is big enough
- **Number of active processes** limited by **number of partitions**
- **Size of processes** limited by the **size of** the largest **partition**
- Wastes memory due to **internal fragmentation**

Dynamic

- Processes are **dynamically allocated** space in main memory
- **Number of active processes** is only limited by the size of the memory
- **Size of a process** is limited by the **size of the memory**
- Wastes memory due to **external fragmentation**

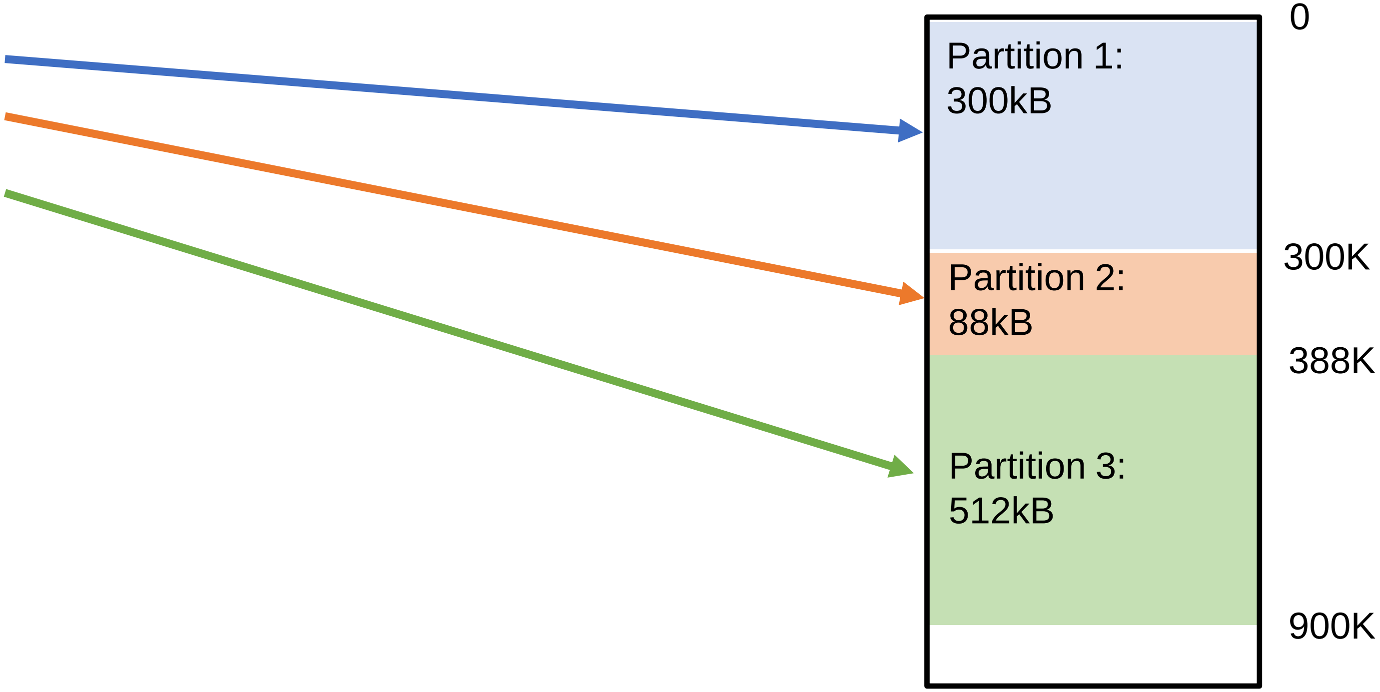
Relocatable

- Same as dynamic, but **can compact memory** to make space for more processes

Relocatable partitioning: example

Assume a main memory of 1MB as shown below, and a process to partition mapping using **Best Fit**.
Consider the following **sequence of process arrivals and leavings**, and the size of their memory address space.

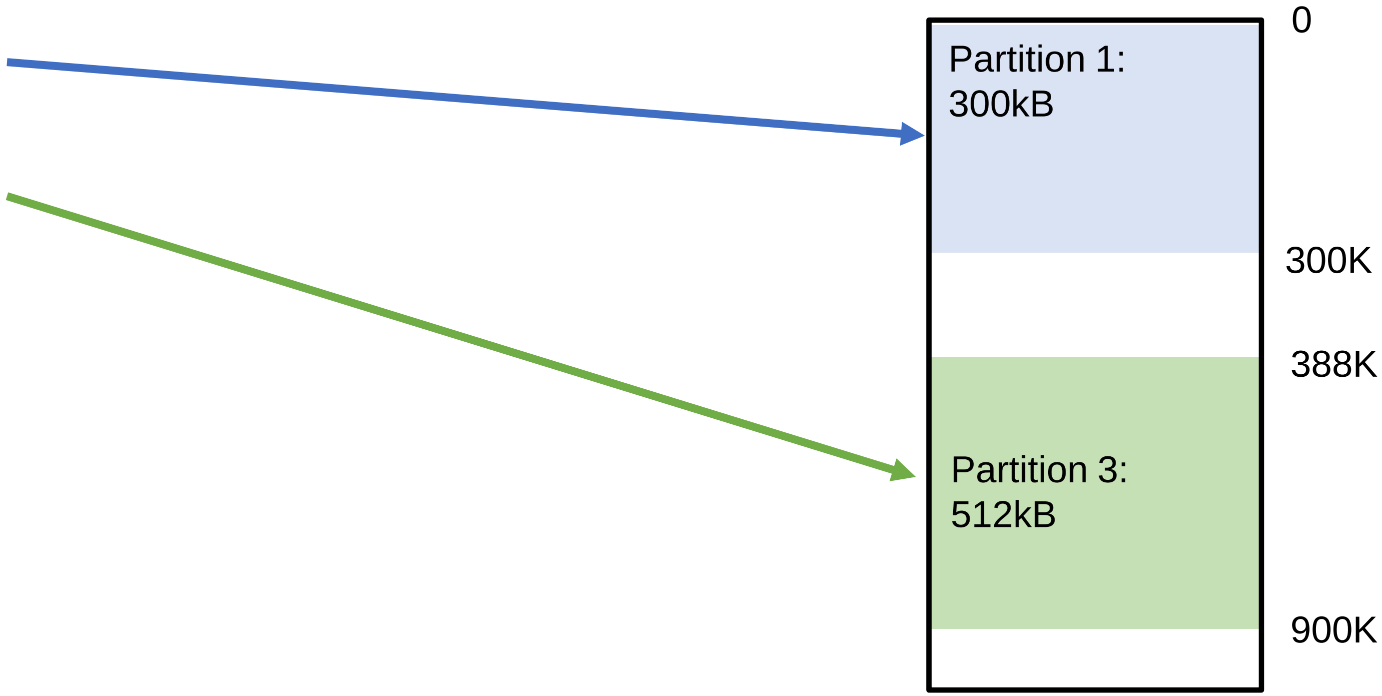
- P1: 300kB
- P2: 88kB
- P3: 512kB
- P2 leaves
- P4: 150kB



Relocatable partitioning: example

Assume a main memory of 1MB as shown below, and a process to partition mapping using **Best Fit**.
Consider the following **sequence of process arrivals and leavings**, and the size of their memory address space.

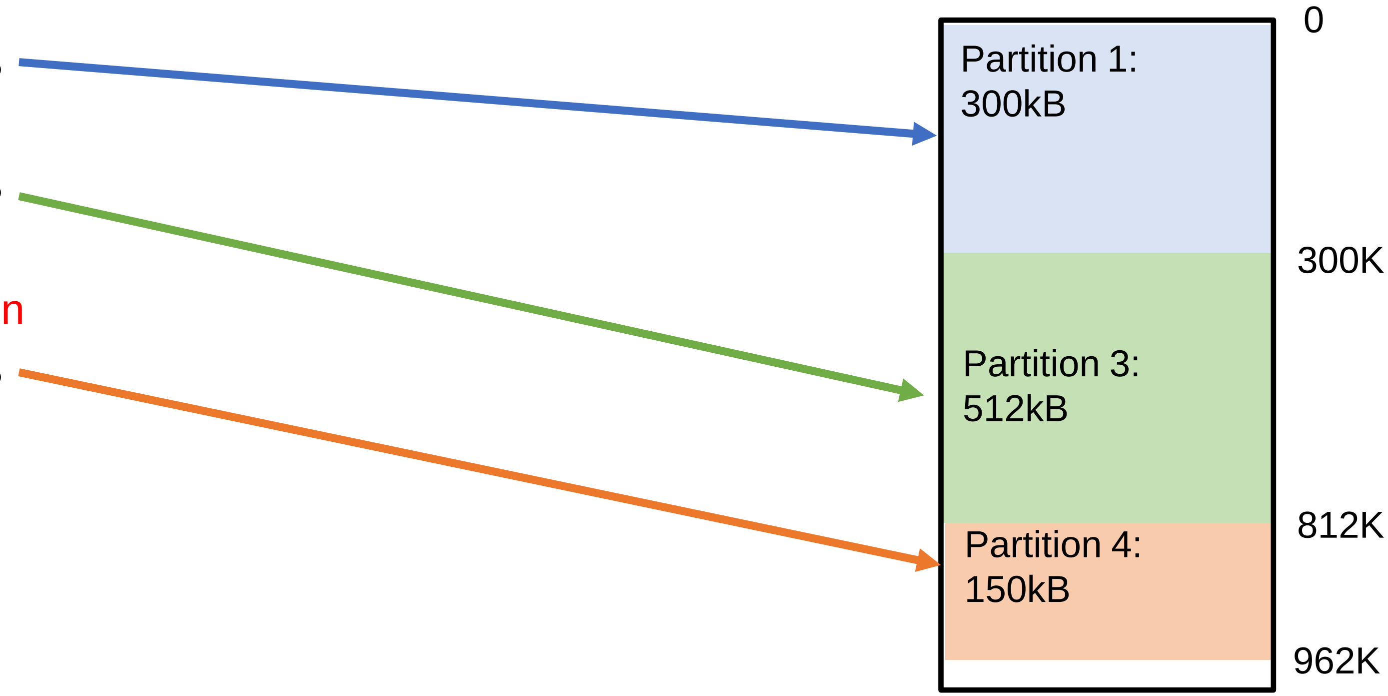
- P1: 300kB
- P2: 88kB
- P3: 512kB
- P2 leaves
- P4: 150kB



Relocatable partitioning: example

Assume a main memory of 1MB as shown below, and a process to partition mapping using **Best Fit**.
Consider the following **sequence of process arrivals and leavings**, and the size of their memory address space.

- P1: 300kB
- P2: 88kB
- P3: 512kB
- P2 leaves
- compaction
- P4: 150kB



Memory management in early systems

- Every active process resides **in its entirety in main memory**
  - **Large programs cannot execute**
- Every active process is **allocated a contiguous part of main memory**.

See preparatory material

- Three **partitioning schemes**

Fixed

- **Partitions** have **fixed size**
- Processes are **assigned to a free partition** that is big enough
- **Number of active processes** limited by **number of partitions**
- **Size of processes** limited by the **size of** the largest **partition**
- Wastes memory due to **internal fragmentation**

Dynamic

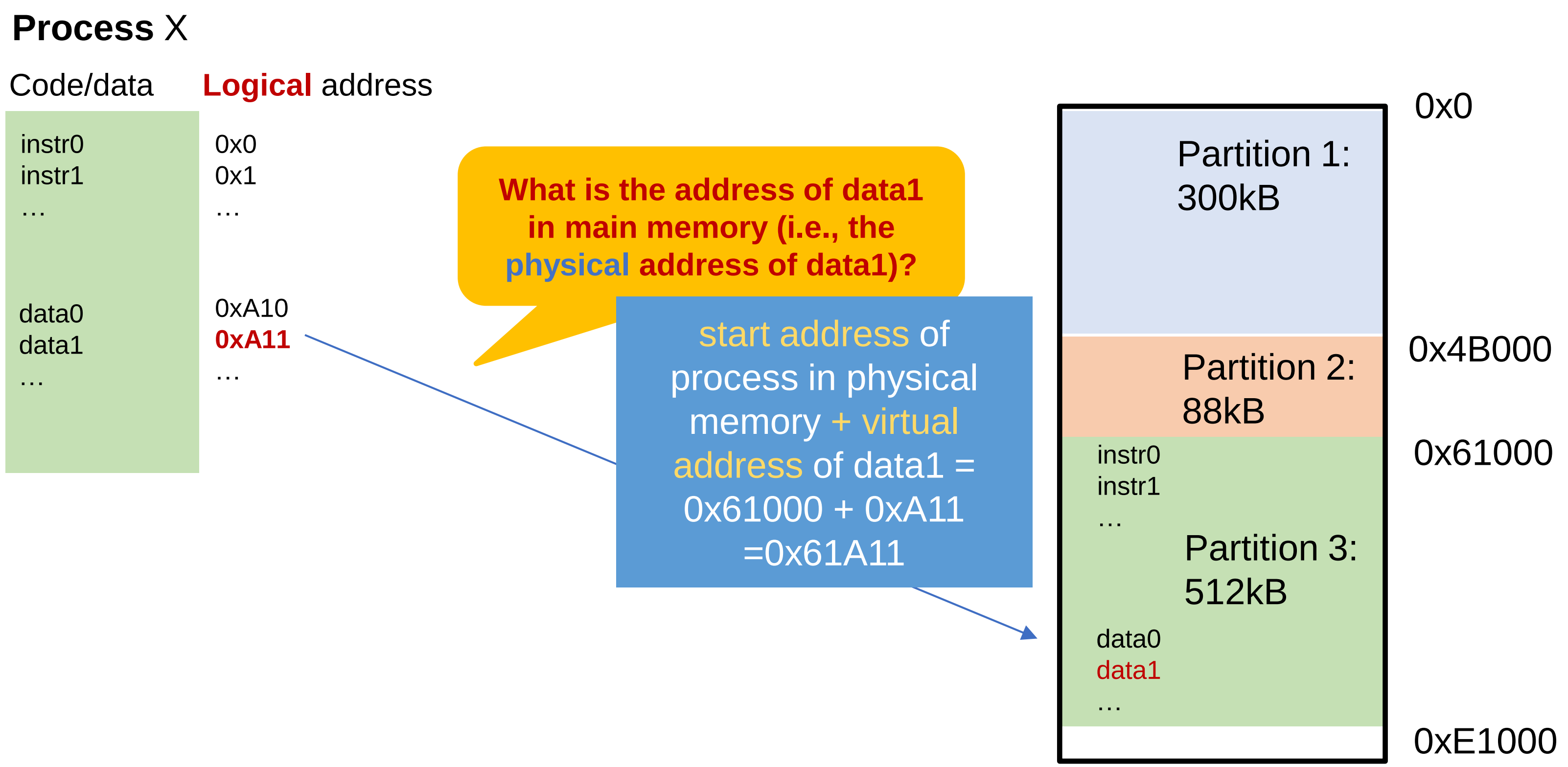
- Processes are **dynamically allocated** space in main memory
- **Number of active processes** is only limited by the size of the memory
- **Size of a process** is limited by the **size of the memory**
- Wastes memory due to **external fragmentation**

Relocatable

- Same as dynamic, but **can compact memory** to make space for more processes
- Compaction is a **costly operation**
- Requires **dynamic address binding**
- **Different compaction algorithms exist.**
  They're explained in the preparatory material

Preparatory material is part of the exam!

Translating virtual addresses in physical addresses (early systems)



Do we have the ideal memory properties so far?

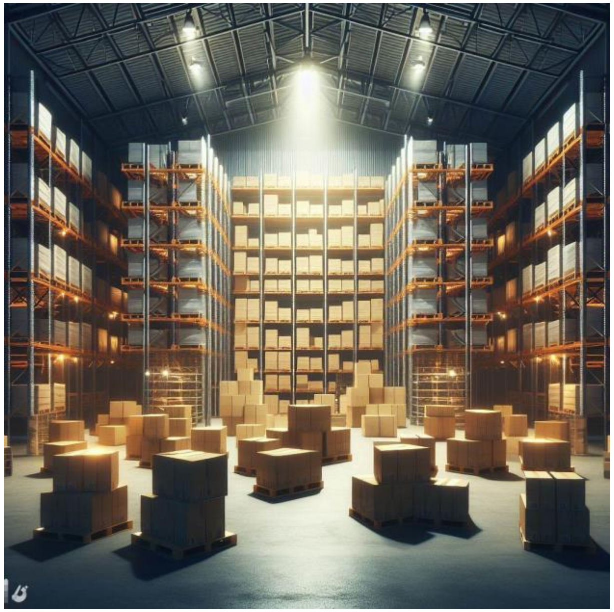
Simple	Yes	
Private	Yes/No	We have **isolation** but **no sharing**
Permanent	No	Unless the programmer enforces a process stays in memory
Fast	No	Process-based **swapping is expensive** (needs to move a complete process in and out) **Compaction is expensive**
Huge	No	Process size cannot exceed physical size of the main memory
Cost-effective	Yes	Using memory hierarchy

Agenda

- Reminder
- Memory paging
- Conclusion

Let’s use an analogy

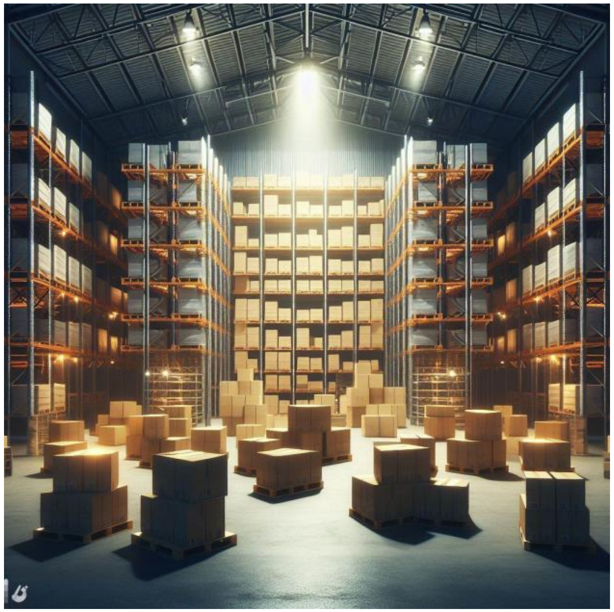
Warehouse (secondary storage)



Office (primary storage)

Let's use an analogy

Warehouse (secondary storage)



A process moved to the office

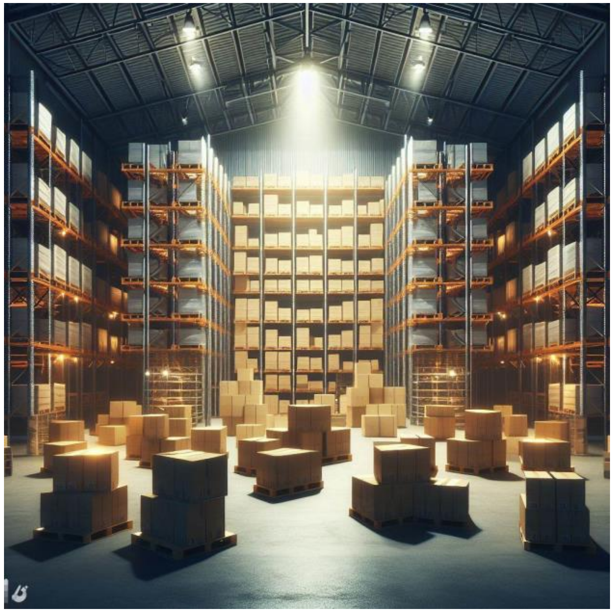
Office (primary storage)

Old (memory) management systems, **move a full pallet** (process) **in the office** whenever we need **to access a single a file** (data/instruction) from one of the boxes

- Long loading time
- Waste of office space
- Few pallets may be accessed at the same time in the office
  ➔must often move a pallet back in the warehouse to access a new pallet (i.e, run another process)

Let’s use an analogy

Warehouse (secondary storage)

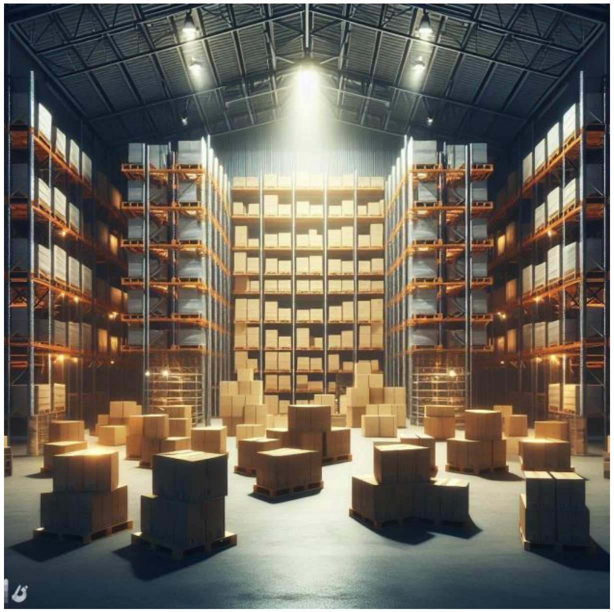


Office (primary storage)

Let's use an analogy

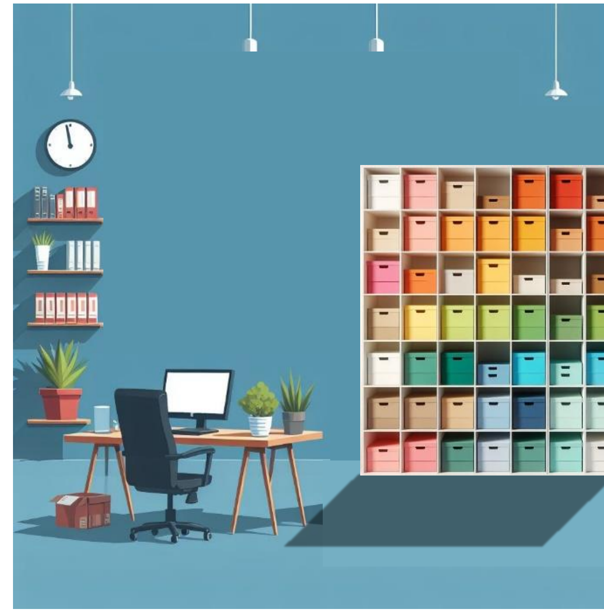
Why are we not moving files instead of boxes?

Warehouse (secondary storage)



Move one box to the office

Office (primary storage)



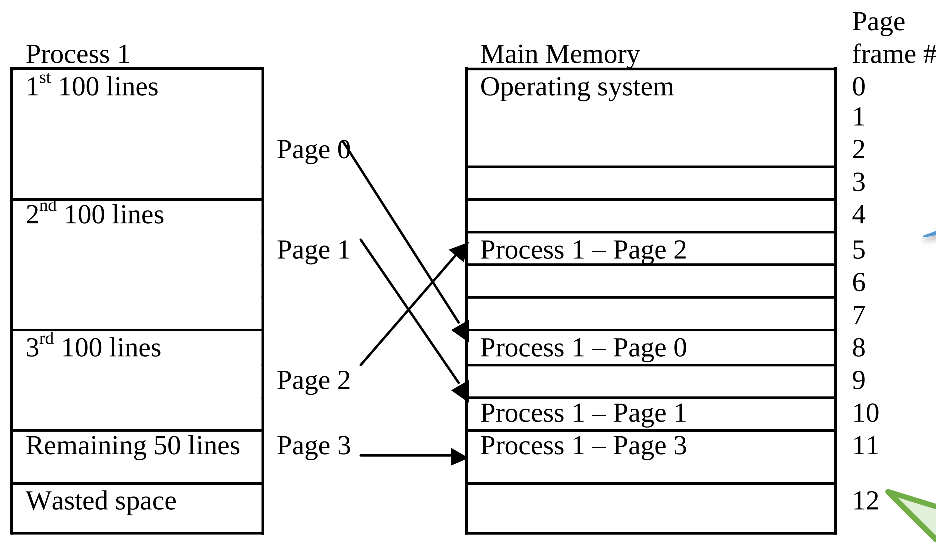
New (memory) management systems, **move one** **(or a set of)** **box** at a time **in the office** whenever we need **to access a file** in that box.

Store boxes on shelves of the same size than a box.

- Short loading time (i.e., quick to start executing a new task)
- Efficient space usage
- Many more concurrent "tasks" running in the office
- Quick to swap boxes

Memory paging

Divide the process address space in **segments of identical sizes called pages**



Divide the main memory in **segments of identical sizes called pages frames**

Advantage: Every free frame is always usable ➔ No external fragmentation

The size of page = the size of a page frame

A page can be loaded in any free page frame (one page per frame)

➔ **programs do not have to be stored contiguously in main memory**

➔ **No external fragmentation**

➔ **Internal fragmentation** is limited to the **size of one page**

➔ **We only need to load the pages we actually use** instead of the whole process

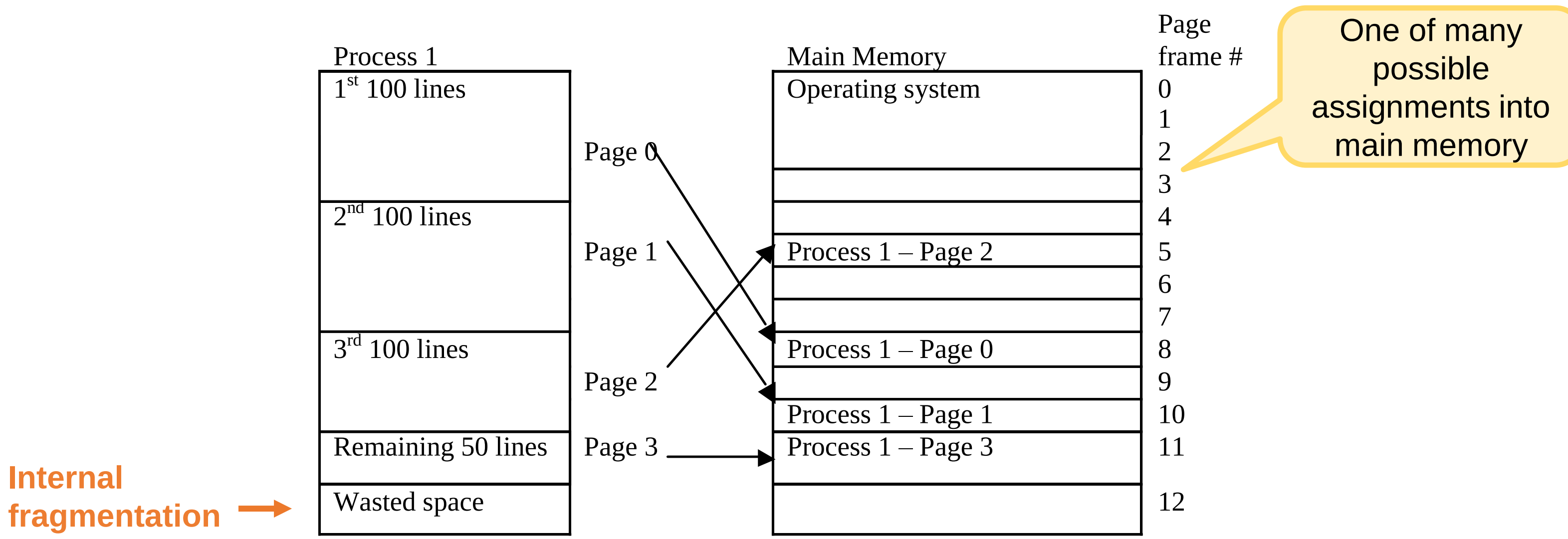
Memory paging: example

Assume that the **main memory** has a **size** of **1300 lines**. Each **page** and each **frame** has a **size** of **100 lines**.

The **OS** memory space **requires 280 lines**, and a loaded **process** P1 has a size of **350 lines**

How many free frames are left in main memory?

There are 13 frames in main memory. OS needs 3 frames, P1 needs 4 => 6 frames left

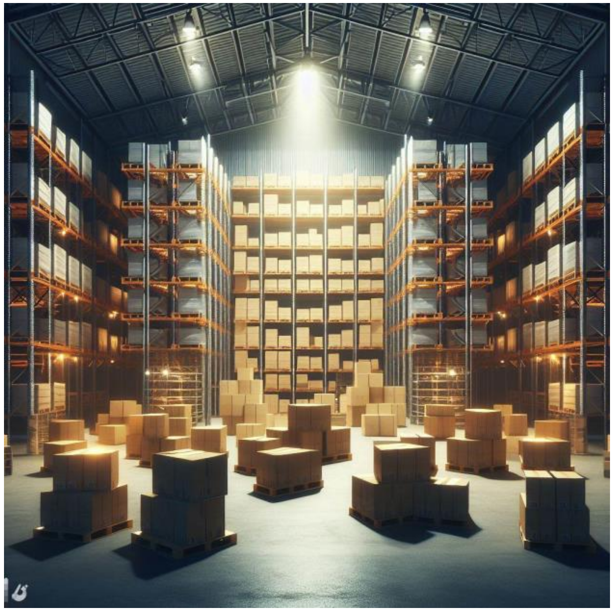


Remaining problems:

1. We must **keep track of where pages are stored**
2. We must provide a mechanism for **address binding** (i.e., translate logical/virtual addresses into physical addresses)
3. How to decide **which page to load** in main memory and when?

Problem 1: keep track of pages (boxes) location

Warehouse (secondary storage)



Office (primary storage)

What solution would you propose to remember which box of which pallet is on which shelf in the office?

Solution 1:

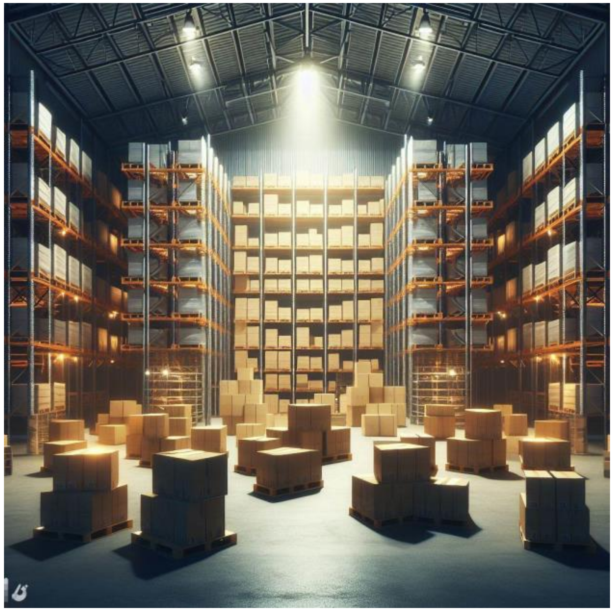
For each pallet in the warehouse, keep track in a ledger **on which shelf each box is** located

Solution 2:

For each shelf in the office, keep track in a ledger **which box of which pallet** is stored there

Problem 1: keep track of pages (boxes) location

Warehouse (secondary storage)



Office (primary storage)

What solution would you propose to remember which box of which pallet is on which shelf in the office?

Solution 1:

For each pallet in the warehouse, keep track in a ledger **on which shelf each box is** located

Called a **page tables** in an OS

Solution 2:

For each shelf in the office, keep track in a ledger **which box of which pallet** is stored the

Called a **frame table** in an OS

Problem 1: keep track of pages location

Solution 1

- The OS maintains a **page table for each process**
- The address at which the page table is saved is recorded in the processes' **PCBs**

Process 0

Page ID	Frame ID
0	2
1	7
2	-
3	5

Process 1

Page ID	Frame ID
0	1
1	8
2	-
3	14
4	4
5	-
6	15

…

Process n

Page ID	Frame ID
0	-
1	0
2	-

Some pages may not be in main memory yet ➔ no frame ID recorded

Records the **frame used by each page** of that process

Advantage:
- **Easy to find a page** in main memory (requires a **single look-up in the page table**)

Disadvantage:
- **Consumes a lot of memory** if each process is made of tens of thousands of pages

Problem 1: keep track of pages location

- **Alternative to page tables (Solution 2)**:
  the OS maintains a single **inverted page table** **(also called frame table)**

Frame ID	Process ID	Page ID
0	3	1
1	2	8
2	-	-
3	1	14
4	0	4
5	2	6
6	3	15
7	-	-

Some frames may not be used yet ➔ no page ID recorded

Records **which page of which process** is loaded **in which frame**

Advantage:
- **Memory consumption** is constant and independent from the number of processes or their size

Disadvantage:
- **Time consuming to find the location of a page** in main memory (requires **up to as many look-ups as there are frames** in the main memory)

Exercise

Consider mapping the virtual memory of a process addressing 1GiB onto a physical memory organized in 256 page frames of 8KiB.

What is the physical memory size?

The virtual memory is made of how many pages?
(assume all virtual memory addresses are consecutive)

How many entries is there in the page table?

What is the size (in bytes) of the page table?

On how many pages is the page table encoded?

Page table: records the **frame used by** each **page** of the process

Exercise

Consider mapping a virtual memory of a process addressing 1GiB onto a physical memory organized in 256 page frames of 8KiB.

What is the physical memory size?

256 frames of 8KiB ➔ physical memory is 256*8KiB=2MiB

The virtual memory is made of how many pages? (assume all virtual memory addresses are consecutive)

1GiB of virtual memory divided in pages of 8KiB ➔ 1GiB/8KiB = 2^{17} pages

How many entries is there in the page table?

As many as the number of pages ➔ 2^{17} entries

What is the size (in bytes) of the page table?

Each entry in the page table encodes the page frame in which the corresponding page is loaded. Since we have 256 page frames, we need 8 bits to identify one page frame ➔ each entry in the page table contains 8 bits ➔ the page table is 2^{17} B = 128KiB

On how many pages is the page table encoded?

128KiB / 8KiB = 16 pages

Exercise

Consider every process addresses a virtual memory of 1GiB (per process). Consider a physical memory organized in 256 page frames of 8KiB. Assume that there are a maximum of 128 processes in the system.

How many entries is there in the frame table (inverted page table)?

What is the size (in bytes) of the frame table?

On how many pages is the frame table encoded?

Frame table: records **which page of which process** is loaded **in which frame**

Exercise

Consider every process addresses a virtual memory of 1GiB (per process).
Consider a physical memory organized in 256 page frames of 8KiB.
Assume that there are a maximum of 128 processes in the system.

Physical memory size = 2MiB
Virtual memory of each process is organized in 2^17 pages

How many entries is there in the frame table?

As many as the number of page frames → 256

What is the size (in bytes) of the frame table?

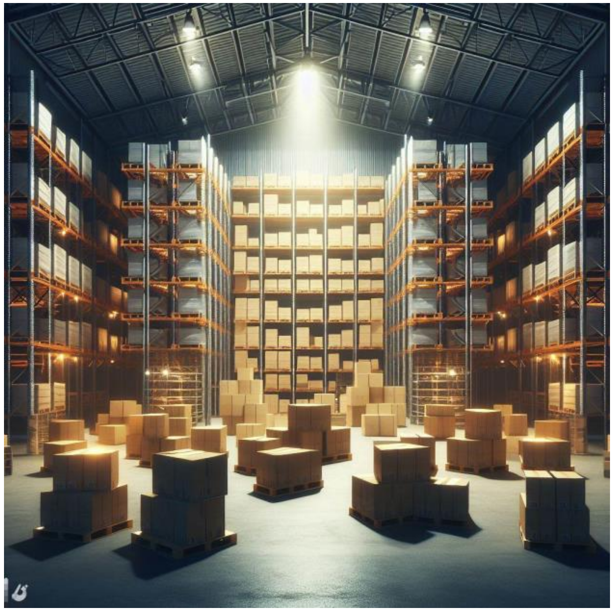
Each entry in the frame table encodes which page from which process is loaded in the corresponding page frame. Since we have 128 processes, we need 7 bits to identify the process. Since we have 2^17 pages in each process, we need 17 bits to identify the page loaded in the frame → each entry needs 7+17=24bits=3 bytes.
→the frame table is 256*3bytes = 768B

On how many pages is the page table encoded?

Since a page has a size of 8KiB, we need a single page to encode the frame table

Problem 2: address binding

Warehouse (secondary storage)



Office (primary storage)

Say I have a ledger of where boxes are stored (page or frame table).
How can I find file 56 in box 23 of pallet 48 in the office?

virtual address of the file on the pallet

Pallet number	Box number	File number

physical address of the file in the office

Shelf number	File number

Problem 2: address binding

Solution

- Use **virtual addresses** in the program code
- **Translate** virtual addresses in **physical addresses** **during the execution** of the code

Virtual address

- Can refer to $2^{|p|}$ pages
- Page size is $2^{|w|}$ words/page
- **Virtual memory size: $2^{|p|+|w|}$ words**

offset in the page in number of words

$\|p\|$ bits	$\|w\|$ bits
Page number p	offset

Virtual memory size seen by each process can be **larger or smaller than** the **physical memory** size

Physical address

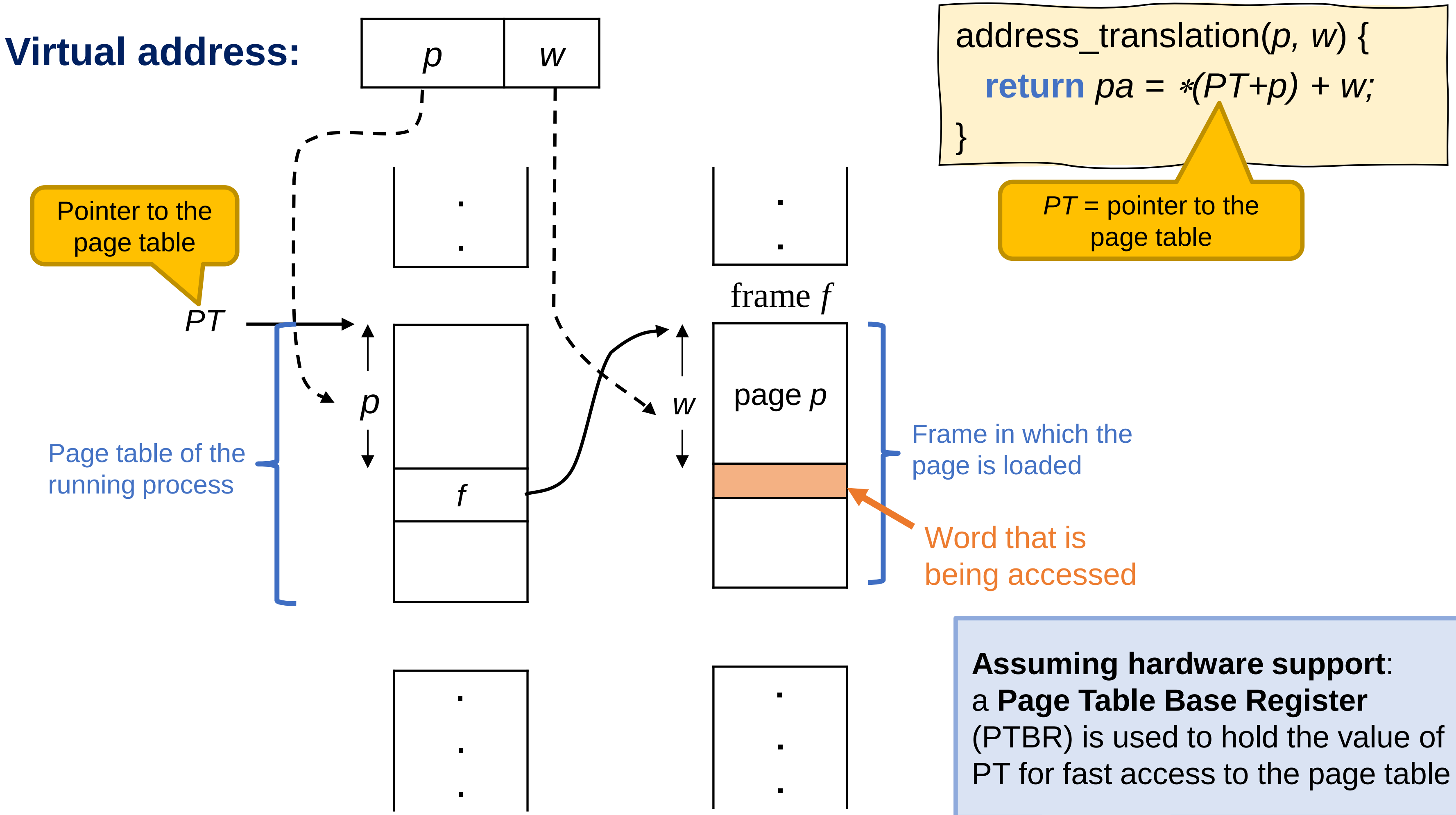
- Can refer to $2^{|f|}$ frames
- Frame size is $2^{|w|}$ words/frame
- **Physical memory size: $2^{|f|+|w|}$ words**

Frame number in which the page we want to access is loaded

$\|f\|$ bits	$\|w\|$ bits
Frame number f	offset

Same as the offset in the virtual address

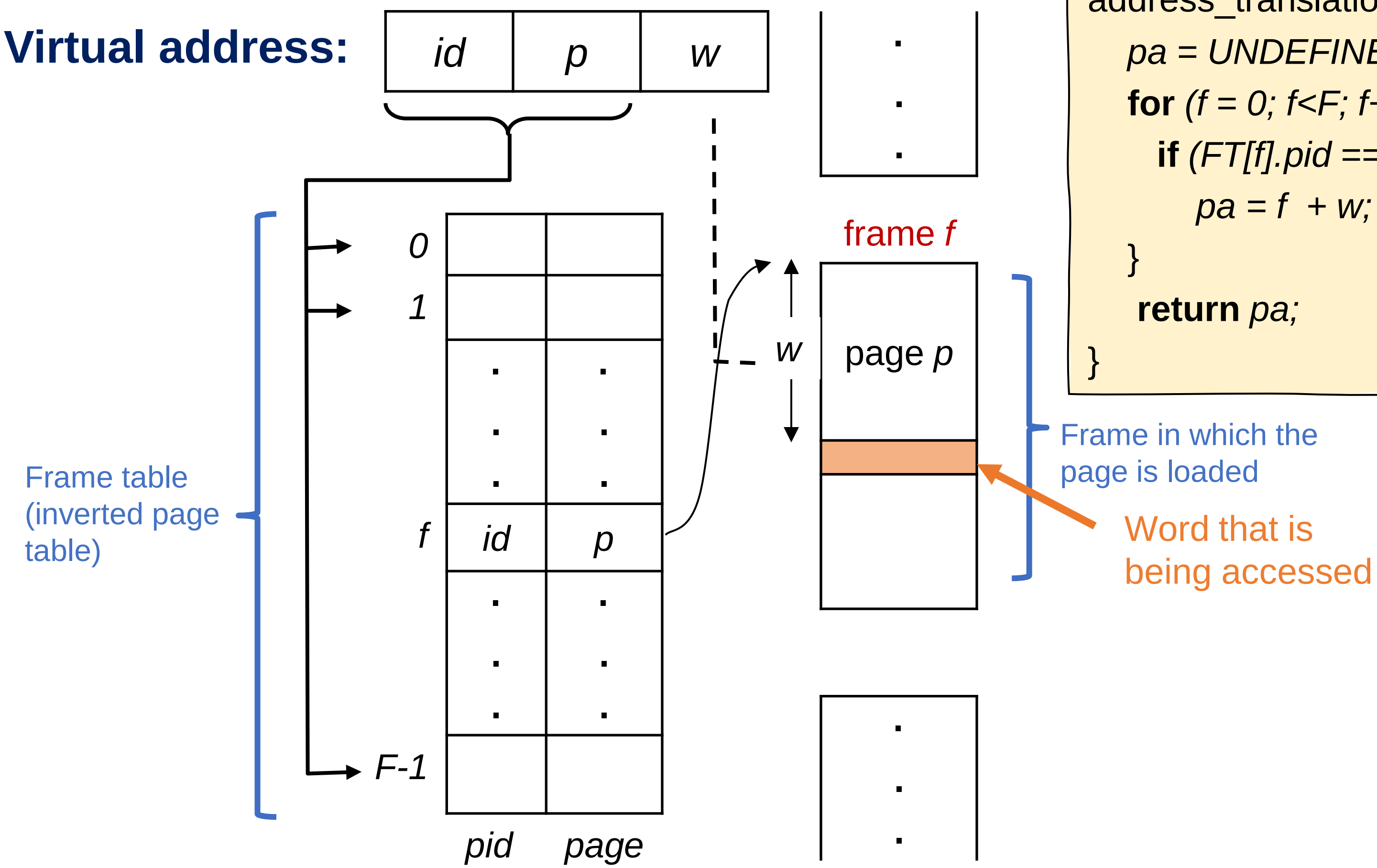
Problem 2: address binding using page tables



Assuming hardware support:
a **Page Table Base Register** (PTBR) is used to hold the value of PT for fast access to the page table

Reading/writing from/to memory requires **two memory accesses**:
one for **accessing the page table**, and one for **accessing actual data**

Problem 2: address binding using the frame table




```
address_translation(id, p, w) {
    pa = UNDEFINED;
    for (f = 0; f<F; f++) {
      if (FT[f].pid == id && FT[f].page == p )
        pa = f + w;
    }
    return pa;
}
```

FT = pointer to the frame table

Reading/writing from/to memory requires up to **as many memory accesses as there are entries in the frame table**

Exercise

Consider a **page table** with 8 entries as shown below. Each page has a size of 256 bytes. The smallest addressable unit is a byte.

	Page table
0	0x5AC0
1	-
2	-
3	0xFFFF
4	0x0012
5	-
6	0x1234
7	-

What is the size of the virtual memory of the process?

What is the size of the physical memory?

On how many bits is the offset of a word encoded?

What is the physical address for the four following virtual addresses?
0x056
0x434
0x500
0x912

Page table: records the page **frame used by** each **page** of the process

Exercise

Consider a **page table** with 8 entries as shown below. Each page has a size of 256 bytes. The smallest addressable unit is a byte.

	Page table
0	0x5AC0
1	-
2	-
3	0xFFFF
4	0x0012
5	-
6	0x1234
7	-

What is the size of the virtual memory of the process?

8 pages of 256B ➔ 8*256B = 2KiB

What is the size of the physical memory?

We see that there is a page frame with ID 0xFFFF
➔ There are at least 2^16 frames, each of size 256B
➔ The physical memory is **at least** 2^16 * 256B = 2^24B = 16MiB

On how many bits is the offset of a word encoded?

Pages are 256B and we address bytes ➔ 256 addresses ➔ we use 8 bits for the address of a word in a page

What is the physical address for the four following virtual addresses?

0x056	Physical address: 0x5AC056
0x434	Physical address: 0x001234
0x500	Undefined. The page must first be loaded in main memory
0x912	Error. Virtual address is outside the address space of the process

Exercise

Consider a **frame table** with 8 entries as shown below. Each page has a size of 4KiB. The smallest addressable unit is a byte.

	Process ID	Page ID
0	5	0x90
1	-	
2	2	0x22
3	2	0x21
4	5	0x80
5	-	
6	1	0x00
7	2	0x04

What is the size of the physical memory?

On how many bits is the offset of a word encoded?

What is the physical address for the four following virtual addresses **of process 2**?

0x21650

0x80123

0x4341

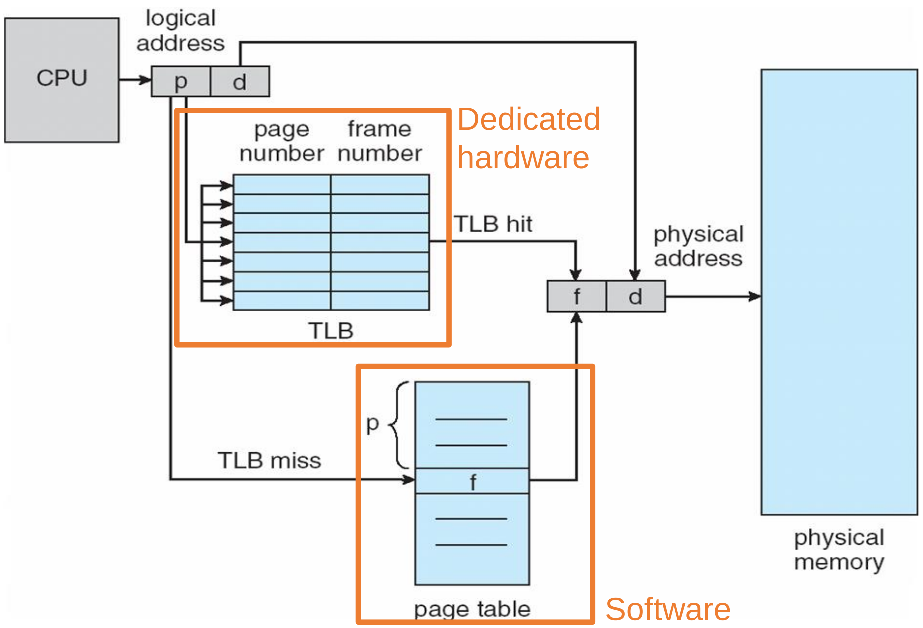
0x2022

Frame table: records **which page of which process** is loaded **in which frame**

Problem 2: address binding (acceleration)

TU/e

- A **Translation Look-aside Buffer** (**TLB**) is a small cache memory in the processor that **keeps track of the locations of the most recently used pages** in main memory
- It **accelerates address translation** and thus memory accesses
- It **does not contain data or instructions**, only the frame id in which the most recently accessed pages are loaded



Source: Fig 9.12, Silberschatz, Galvin, Gagne:Operating System Concepts, 10th Edition, Global edition

- **Whenever a page is accessed**
  - First, check if it is in the TLB
    - **if** the page location **cannot be found in the TLB**, **use the page table**
    - if the page location still cannot be found a **page fault** is generated (i.e., the page is not in physical memory)

IRiS

Problem 3: How to decide which page to load and when?

Use **demand paging**:

- Bring a **page into main memory** **only when it is accessed for the first time**
  - No need to have the entire process stored in memory

Takes advantage of the fact that **not all pages are necessary at once**
Examples:
User-written **error handling code**
Mutually exclusive modules/segments of code
Only **fractions of large tables** are actually used

More problems:
- **For how long** should a page stay in main memory
- **How many pages** of each process should be kept in main memory
- What to do when there is no more free frames?

See next lecture.

Summary

TU/e

- The **memory management subsystem** **decides what to load, where and when in main memory**
- **Early systems** required to **load entire processes in contiguous memory regions**
  - Slow
  - Limited concurrency level
  - Fragmentation (internal and external)
- **Memory paging eliminates external fragmentation** and **strongly limits internal fragmentation**
- **Page or Frame tables** allow to **dynamically bind virtual addresses** in the process address space **to physical addresses** in main memory

IRiS

Next lecture

- **Load and replacement strategies** for virtual memory
- A new homework will be released during the weekend, deadline after the vacations